INTERVIEW MINI-BOOK

QUERIES TO OFFERS

SQL

QUICK INTERVIEW EDITION

Interview Guide · From SELECT to Window Functions

The SQL every interview tests, in simple language: joins, GROUP BY, subqueries, CTEs, window functions and 22 classic interview problems solved step by step, with a theory question bank.



interview.sql

```sql
SELECT name, salary
FROM (SELECT name, salary, DENSE_RANK() OVER (ORDER BY salary DESC) AS rnk
      FROM employees) t
WHERE rnk = 2;    -- 2nd highest salary, ties handled
```

SELECT · WHERE · GROUP BY · HAVING · JOINs · Subqueries · CTEs · Window Functions · RANK · LAG / LEAD · Gaps & Islands · Indexes

WRITTEN BY

Shreyash Tripathi

8 Chapters · Interview Q&A · Practical Tasks · Answers

Edition 2026

JOB-READY SERIES · INTERVIEW MINI-BOOK



SQL for Interviews

Quick Interview Edition

Read the question | Pick the pattern | Write the query | Check edge cases

WRITTEN BY

Shreyash Tripathi

Interview Mini-Book, 2026

© Shreyash Tripathi. All rights reserved.

Preface

SQL rounds appear in interviews for developers, analysts, data engineers and product roles, and they keep asking the same things: joins, GROUP BY, subqueries, window functions, and a handful of classic problems like the Nth highest salary and consecutive logins. This small book teaches exactly those, on one shared practice schema you can run yourself.

It is written like a school textbook, only shorter: numbered chapters, simple language, a check with answers after every topic, Interview Corners with the questions you will actually be asked, practical tasks, and a full answer key at the end of every chapter. The last chapter is a complete interview question bank. Read one chapter a day and you will be interview-ready in about a week.

Shreyash Tripathi

Features of Every Chapter

🎯 **INTERVIEW CORNER**

Questions interviewers really ask, with model answers.

🔑 **KEY POINT**

The one thing to remember.

⚠️ **COMMON MISTAKE**

Traps that cost marks in interviews.

🖍️ **TRY IT**

A small task to do yourself right now.

✅ **CHECK YOUR PROGRESS**

Short questions with answers after every topic, plus practice questions and an answer key at the end of each chapter.

Contents

1 SQL Basics and How a Query Runs 5

2 Aggregation, GROUP BY and CASE 15

3 Joins 25

4 Subqueries and CTEs 34

5 Window Functions 44

6 Classic Interview Problems I 53

7 Classic Interview Problems II: Analytics 62

8 SQL Theory, Performance and Question Bank 71

APPENDICES

A SQL Syntax Cheat Sheet 86

B Practice Schema and Sample Data 90

C Glossary 94

CHAPTER 1

SQL Basics and How a Query Runs

*SELECT * FROM life WHERE problems IS NULL; -- 0 rows returned.*

After studying this chapter, you will be able to

- Name the five families of SQL commands and give an example of each.
- Write SELECT queries with WHERE, DISTINCT, ORDER BY, LIMIT and aliases.
- Explain how NULL and three-valued logic change the result of a filter.
- Describe the logical order in which a query runs and why a SELECT alias fails in WHERE.
- Choose correctly between DELETE, TRUNCATE and DROP.

Think of a kirana store owner with a thick notebook: who bought what, on which date, for how much. One day the notebook becomes a computer table, and the owner wants to ask "Who spent more than 1,000 rupees last month?" SQL is the language for asking that question. Almost every interview for analyst, developer or data roles starts here, so let us build a strong base.

1.1 What SQL Is and Its Five Command Families

SQL (Structured Query Language) is the standard language for working with relational databases, where data lives in tables of rows and columns. SQL is **declarative**: you say *what* you want, and the database engine decides *how* to fetch it. PostgreSQL, MySQL, SQL Server, Oracle and SQLite all speak SQL, each with small dialect differences.

SQL commands are grouped into five families:

Family	Full form	Purpose	Commands
DDL	Data Definition Language	Define or change structure	CREATE, ALTER, DROP, TRUNCATE, RENAME
DML	Data Manipulation Language	Change the rows	INSERT, UPDATE, DELETE, MERGE
DQL	Data Query Language	Read data	SELECT
DCL	Data Control Language	Permissions	GRANT, REVOKE
TCL	Transaction Control Language	Manage transactions	COMMIT, ROLLBACK, SAVEPOINT

Some books put SELECT inside DML. Both views are accepted; just say so if asked.

DELETE vs TRUNCATE vs DROP

All three "remove" something, but very differently.

```
DELETE FROM orders WHERE status = 'cancelled';  -- removes chosen rows
TRUNCATE TABLE logins;                          -- removes all rows, keeps table
DROP TABLE logins;                              -- removes the table itself
```

Point	DELETE	TRUNCATE	DROP
Family	DML	DDL	DDL
WHERE allowed?	Yes	No	No
Structure kept?	Yes	Yes	No
Speed on big tables	Slow (row by row, logged)	Fast (deallocates pages)	Fast
Fires DELETE triggers?	Yes	No	No
Rollback?	Yes, inside a transaction	Depends on database	Depends on database

Version note: in PostgreSQL and SQL Server, TRUNCATE and DROP can be rolled back inside a transaction. In MySQL and Oracle they cause an implicit commit, so there is no going back. TRUNCATE resets the auto-increment counter in MySQL; in PostgreSQL write `TRUNCATE TABLE t RESTART IDENTITY`.

> 🎯 **INTERVIEW CORNER**
>
> **Q:** Why is TRUNCATE faster than DELETE without a WHERE clause? **A:** DELETE removes and logs each row one at a time and fires triggers. TRUNCATE simply deallocates the data pages, logs very little and skips row triggers.

> ✅ **CHECK YOUR PROGRESS 1.1**
>
> 1. **Which family does GRANT belong to?**
>
>   **Answer:** DCL (Data Control Language), along with REVOKE.
>
> 2. **You must remove only cancelled orders. DELETE or TRUNCATE?**
>
>   **Answer:** DELETE, because only DELETE accepts a WHERE clause.

1.2 The Sample Schema Used in This Book

Every chapter uses the same five tables, so you can focus on SQL rather than new data each time. Here are the DDL statements. Parent tables are created first so that foreign keys have something to point to.

```
CREATE TABLE departments (
  dept_id    INT PRIMARY KEY,
  dept_name  VARCHAR(50) NOT NULL
);

CREATE TABLE employees (
  emp_id     INT PRIMARY KEY,
  name       VARCHAR(100) NOT NULL,
  dept_id    INT REFERENCES departments(dept_id),
  manager_id INT REFERENCES employees(emp_id),   -- self reference
  salary     DECIMAL(10,2),
  hire_date  DATE
);

CREATE TABLE customers (
  customer_id INT PRIMARY KEY,
  name        VARCHAR(100) NOT NULL,
  city        VARCHAR(50),
  signup_date DATE
);

CREATE TABLE orders (
  order_id    INT PRIMARY KEY,
  customer_id INT REFERENCES customers(customer_id),
  order_date  DATE NOT NULL,
  amount      DECIMAL(10,2),
  status      VARCHAR(20)        -- 'delivered', 'pending', 'cancelled'
);

CREATE TABLE logins (
  user_id    INT,                -- same as customers.customer_id
  login_date DATE
);
```

Sample rows we will use in this chapter:

emp_id	name	dept_id	manager_id	salary	hire_date
1	Asha	10	NULL	90000	2019-04-01
2	Ravi	10	1	60000	2021-06-15
3	Meena	20	1	55000	2022-01-10
4	Karan	20	3	48000	2023-03-05
5	Priya	NULL	1	52000	2024-07-20

Departments: 10 = Sales, 20 = Finance, 30 = HR. Note that Asha has no manager (she is the boss) and Priya has not yet been assigned a department. Those NULLs will teach us a lot.

> 🔑 **KEY POINT**
>
> A PRIMARY KEY is unique and NOT NULL. A FOREIGN KEY (REFERENCES) must match a value in the parent table or be NULL. In MySQL, foreign keys are enforced only with the InnoDB engine (the default since 5.5).

CHECK YOUR PROGRESS 1.2

1. **Why must departments be created before employees?**
   Answer: employees.dept_id references departments, and a foreign key cannot point to a table that does not exist yet.
2. **What does manager_id REFERENCES employees(emp_id) model?**
   Answer: A self-referencing hierarchy: every manager is also an employee in the same table.

1.3 SELECT, WHERE, DISTINCT and Aliases

The simplest query picks columns from a table. Avoid `SELECT *` in real code; list the columns you need.

```
SELECT name, salary
FROM employees
WHERE dept_id IN (10, 20)
  AND salary BETWEEN 50000 AND 90000;
```

name	salary
Asha	90000
Ravi	60000
Meena	55000

BETWEEN is **inclusive** at both ends, so Asha (exactly 90000) is in. Karan fails the salary test, and Priya fails because NULL IN (10, 20) is not true.

Common WHERE operators

Operator	Example	Meaning
=, <> (or !=), <, >, <=, >=	salary >= 55000	Comparison
IN / NOT IN	city IN ('Pune', 'Delhi')	Matches any value in a list
BETWEEN a AND b	order_date BETWEEN '2024-01-01' AND '2024-03-31'	Range, inclusive
LIKE	name LIKE 'A%'	% = any characters, _ = exactly one
IS NULL / IS NOT NULL	manager_id IS NULL	Tests for missing value
AND, OR, NOT	a AND (b OR c)	Combine conditions

`WHERE name LIKE '_a%'` finds names whose second letter is "a": Ravi and Karan. **Version note:** LIKE is case-sensitive in PostgreSQL (use `ILIKE` for case-insensitive) but usually case-insensitive in MySQL because of its default collation. AND is evaluated before OR, so use brackets when you mix them.

DISTINCT removes duplicate rows from the result:

```
SELECT DISTINCT dept_id FROM employees;
```

dept_id
10
20
NULL

DISTINCT treats all NULLs as one group, so you get one NULL row. DISTINCT applies to the whole row, not one column: `SELECT DISTINCT dept_id, name` returns all five employees.

Aliases rename a column or table for the query only. AS is optional for columns; Oracle does not allow AS before a table alias.

```
SELECT e.name AS employee, e.salary * 12 AS annual_salary
FROM employees e;
```

Use double quotes for aliases with spaces (`"Annual Salary"`) in standard SQL and PostgreSQL; MySQL also accepts backticks.

> 🏏 **TRY IT**
>
> Write a query to list customers from Mumbai or Bengaluru whose name starts with "S". Then rewrite it with OR instead of IN and check that you used brackets.

✅ **CHECK YOUR PROGRESS 1.3**

1. **Does `salary BETWEEN 50000 AND 60000` include 60000?**
   Answer: Yes. BETWEEN includes both boundary values.
2. **Which pattern finds names ending in "a"?**
   Answer: `name LIKE '%a'` .

1.4 ORDER BY, LIMIT and OFFSET

A table has no guaranteed order. If you need sorted output, you **must** write ORDER BY. ASC is the default; DESC sorts largest first. You can sort by many columns.

```
SELECT name, salary
FROM employees
ORDER BY salary DESC
LIMIT 2 OFFSET 1;     -- skip the top earner, take the next 2
```

name	salary
Ravi	60000
Meena	55000

Different databases write "first N rows" differently:

Database	Syntax
PostgreSQL, MySQL, SQLite	LIMIT 2 OFFSET 1 (MySQL also: LIMIT 1, 2)
SQL Server	SELECT TOP 2 ... or OFFSET 1 ROWS FETCH NEXT 2 ROWS ONLY
Standard SQL, Oracle 12c+, PostgreSQL	OFFSET 1 ROWS FETCH FIRST 2 ROWS ONLY

In SQL Server, OFFSET...FETCH needs an ORDER BY. Careful in MySQL: `LIMIT 1, 2` means offset 1, count 2, which is the reverse of what many people guess.

NULLs in sorting: PostgreSQL and Oracle put NULLs last in ASC order; MySQL and SQL Server put them first. PostgreSQL and Oracle let you choose with `NULLS FIRST` or `NULLS LAST`.

> 😄 **FUNNY SIDE**
>
> LIMIT without ORDER BY is like asking "give me any two players from the team" and then complaining that Virat was not picked. The database never promised you Virat.

> ✅ **CHECK YOUR PROGRESS 1.4**
>
> 1. **Write a query for the third highest salary row (assume no ties) in PostgreSQL.**
>
>   **Answer:** `SELECT name, salary FROM employees ORDER BY salary DESC LIMIT 1 OFFSET 2;`
>
> 2. **Is the order of rows guaranteed without ORDER BY?**
>
>   **Answer:** No. It may look stable, but it can change with indexes, parallel plans or new data.

1.5 NULL and Three-Valued Logic

NULL means "unknown" or "missing". It is not zero and not an empty string. Any comparison with NULL, even `NULL = NULL`, gives **UNKNOWN**, not TRUE. SQL therefore uses three truth values: TRUE, FALSE and UNKNOWN. WHERE keeps a row only when the condition is TRUE.

Expression	Result
TRUE AND UNKNOWN	UNKNOWN
FALSE AND UNKNOWN	FALSE
TRUE OR UNKNOWN	TRUE
FALSE OR UNKNOWN	UNKNOWN
NOT UNKNOWN	UNKNOWN

Watch what happens to Priya:

```
SELECT name FROM employees WHERE dept_id <> 10;
```

name
Meena
Karan

Priya is not in department 10, yet she is missing, because `NULL <> 10` is UNKNOWN. To include her, write `WHERE dept_id <> 10 OR dept_id IS NULL`. Use `COALESCE(dept_id, 0)` when you want to replace NULL with a default value.

> ⚠️ **COMMON MISTAKE**
>
> Writing `WHERE manager_id = NULL`. It returns zero rows, always. Use `IS NULL`. A second trap: `WHERE dept_id NOT IN (10, NULL)` also returns zero rows, because `x <> NULL` is UNKNOWN for every row. This often bites when NOT IN uses a subquery whose column contains a NULL.

> 🔑 **KEY POINT**
>
> Aggregates like SUM, AVG and COUNT(column) ignore NULLs, but COUNT(*) counts every row. In our data, COUNT(*) = 5 but COUNT(dept_id) = 4.

> ✅ **CHECK YOUR PROGRESS 1.5**
>
> 1. **What does `NULL = NULL` evaluate to?**
>
>   **Answer:** UNKNOWN, so a WHERE clause treats it like false and drops the row.
>
> 2. **Find employees who have no manager.**
>
>   **Answer:** `SELECT name FROM employees WHERE manager_id IS NULL;` returns Asha.

1.6 How a Query Really Runs

We *write* SELECT first, but the database *processes* the clauses in a different logical order:

Step	Clause	What happens
1	FROM (and JOINs)	Pick the tables and combine them
2	WHERE	Filter individual rows
3	GROUP BY	Form groups
4	HAVING	Filter groups
5	SELECT	Compute columns and aliases (then DISTINCT)
6	ORDER BY	Sort the result
7	LIMIT / OFFSET / FETCH	Cut the rows

This explains a classic error:

```
SELECT name, salary * 12 AS annual
FROM employees
WHERE annual > 700000;     -- ERROR: column "annual" does not exist
```

WHERE runs in step 2, but the alias `annual` is created only in step 5. Fix it by repeating the expression, or by wrapping the query in a subquery or CTE:

```
SELECT name, salary * 12 AS annual
FROM employees
WHERE salary * 12 > 700000
ORDER BY annual DESC;      -- alias is fine here: ORDER BY runs after SELECT
```

name	annual
Asha	1080000
Ravi	720000

Version note: MySQL and PostgreSQL are lenient and also accept SELECT aliases in GROUP BY, and MySQL even in HAVING. This is not standard, so do not rely on it in interviews.

> 💡 **DO YOU KNOW?**
>
> This is the *logical* order. The optimizer may physically do things differently, for example pushing a filter into an index scan, but the result must be the same as if the logical order were followed.

> 🎯 **INTERVIEW CORNER**
>
> **Q:** What is the difference between WHERE and HAVING? **A:** WHERE filters rows before grouping and cannot use aggregates. HAVING filters groups after GROUP BY and can use aggregates such as `HAVING COUNT(*) > 2`.

> 🔑 **KEY POINT**
>
> Remember the order as "**F**ew **W**orkers **G**o **H**ome **S**aturday, **O**ffice **L**ocked": FROM, WHERE, GROUP BY, HAVING, SELECT, ORDER BY, LIMIT.

> ✅ **CHECK YOUR PROGRESS 1.6**
>
> 1. **Why can ORDER BY use a SELECT alias but WHERE cannot?**
>
>   **Answer:** ORDER BY runs after SELECT, so the alias already exists. WHERE runs before SELECT.
> 2. **Which runs first, HAVING or SELECT?**
>
>   **Answer:** HAVING. It filters groups before the SELECT list is computed.

What Have We Learnt?

- SQL is declarative and has five families: DDL, DML, DQL, DCL and TCL.
- DELETE removes chosen rows; TRUNCATE empties the table quickly; DROP removes the table itself.
- WHERE filters rows with operators such as IN, BETWEEN (inclusive), LIKE and IS NULL.
- Only ORDER BY guarantees order; LIMIT/OFFSET, TOP and FETCH vary by database.
- NULL comparisons give UNKNOWN, and WHERE keeps only TRUE rows.
- Logical order: FROM, WHERE, GROUP BY, HAVING, SELECT, ORDER BY, LIMIT. That is why a SELECT alias fails in WHERE.

Practice

A. Multiple Choice Questions

1. Which command belongs to TCL?
   a. GRANT
   b. ROLLBACK
   c. TRUNCATE
   d. SELECT

2. What does `SELECT name FROM employees WHERE dept_id <> 20;` return for our data?
   a. Asha, Ravi, Priya
   b. Asha, Ravi
   c. Meena, Karan
   d. All five

3. In MySQL, `LIMIT 5, 10` means
   a. 5 rows after skipping 10
   b. rows 5 to 10
   c. 10 rows after skipping 5
   d. a syntax error

4. Which clause is logically processed first?
   a. SELECT
   b. ORDER BY
   c. HAVING
   d. WHERE

B. Practical Tasks

1. List delivered orders placed in February 2024 with amount above 1,000, newest first.
2. Show each employee's name and a column `dept` that shows the dept_id, or 0 if it is missing.
3. Return the 2nd and 3rd most recent customer signups, once in PostgreSQL syntax and once in SQL Server syntax.

Answer Key: Chapter 1

A. MCQ

1. (b) ROLLBACK is transaction control; GRANT is DCL, TRUNCATE is DDL.
2. (b) Priya's NULL dept_id gives UNKNOWN, so she is dropped.
3. (c) MySQL's two-argument form is LIMIT offset, count.
4. (d) WHERE runs right after FROM.

B. Practical Tasks

1.
```sql
SELECT order_id, customer_id, order_date, amount
FROM orders
WHERE status = 'delivered'
  AND order_date ⩾ '2024-02-01' AND order_date < '2024-03-01'
  AND amount > 1000
ORDER BY order_date DESC;
```

2.
```sql
SELECT name, COALESCE(dept_id, 0) AS dept
FROM employees;
```

3.
```sql
-- PostgreSQL
SELECT name, signup_date FROM customers
ORDER BY signup_date DESC
LIMIT 2 OFFSET 1;

-- SQL Server
SELECT name, signup_date FROM customers
ORDER BY signup_date DESC
OFFSET 1 ROWS FETCH NEXT 2 ROWS ONLY;
```

CHAPTER 2

Aggregation, GROUP BY and CASE

"My manager asked for one number. Then he asked for it by month, by city and by status."

After studying this chapter, you will be able to

- Use SUM, AVG, MIN, MAX and the three kinds of COUNT, and explain how each treats NULL.
- Write GROUP BY queries correctly and choose between WHERE and HAVING.
- Build labels, buckets and pivot tables with CASE WHEN.
- Use COALESCE, NULLIF and common string and date functions in reports.
- Write the standard report queries asked in analyst interviews.

A kirana store owner does not want 5,000 bill lines. He wants "how much did I sell each month?" and "which items made money?" Turning many rows into a few summary rows is called **aggregation**. Almost every MIS report and half of all SQL interview questions use it. We will use this small `orders` table all through the chapter.

order_id	customer_id	order_date	amount	status
101	1	2024-01-05	500	Delivered
102	2	2024-01-18	1200	Delivered
103	1	2024-02-02	300	Cancelled
104	3	2024-02-10	NULL	Pending
105	2	2024-02-25	800	Delivered
106	4	2024-03-03	450	Cancelled

2.1 Aggregate Functions and NULL

An aggregate function takes many rows and returns one value. The five you must know are COUNT, SUM, AVG, MIN and MAX. The golden rule: **all aggregates except COUNT(*) ignore NULL.**

```
SELECT COUNT(*)                   AS total_rows,
       COUNT(amount)              AS rows_with_amount,
       COUNT(DISTINCT customer_id) AS unique_customers,
       SUM(amount)                AS revenue,
       AVG(amount)                AS avg_amount,
       MIN(order_date)            AS first_order,
       MAX(amount)                AS biggest_order
FROM orders;
```

total_rows	rows_with_amount	unique_customers	revenue	avg_amount	first_order	biggest_order
6	5	4	3250	650	2024-01-05	1200

Look at the average. It is 3250 / 5 = 650, not 3250 / 6. Order 104 has no amount, so AVG simply skips it. If you want NULL to count as zero, say so: `AVG(COALESCE(amount, 0))` gives 541.67.

- **COUNT(*)** counts rows, NULL or not.
- **COUNT(col)** counts rows where col is not NULL.
- **COUNT(DISTINCT col)** counts unique non-NULL values.

On an empty table (or when every value is NULL), COUNT returns 0 but SUM, AVG, MIN and MAX return NULL. MIN and MAX also work on text and dates.

> 🔑 **KEY POINT**
>
> SUM of nothing is NULL, not 0. In reports, write `COALESCE(SUM(amount), 0)` so the boss sees 0 instead of a blank cell.

> ✅ **CHECK YOUR PROGRESS 2.1**
>
> 1. **A column has values 10, NULL, 20. What do COUNT(*), COUNT(col) and AVG(col) return?**
>   **Answer:** 3, 2 and 15.
> 2. **Does COUNT(DISTINCT col) count NULL as a value?**
>   **Answer:** No. NULLs are ignored.

2.2 GROUP BY, HAVING and WHERE

GROUP BY splits rows into buckets and runs the aggregate once per bucket. Revenue per status:

```
SELECT status, COUNT(*) AS orders, SUM(amount) AS revenue
FROM orders
GROUP BY status
ORDER BY revenue DESC;
```

status	orders	revenue
Delivered	3	2500
Cancelled	2	750
Pending	1	NULL

The GROUP BY rule: every column in SELECT must either be in GROUP BY or be inside an aggregate. `SELECT status, order_date, COUNT(*) ... GROUP BY status` is an error, because one status has many dates. MySQL 5.7+ enforces this with `ONLY_FULL_GROUP_BY` (on by default); old MySQL silently picked a random value. PostgreSQL allows extra columns only when you group by the table's primary key.

NULL values form one group of their own. Also, GROUP BY does not promise any order, so always add ORDER BY.

WHERE vs HAVING

WHERE filters rows before grouping. **HAVING** filters groups after grouping, so only HAVING can use aggregates. Customers with 2 or more delivered orders:

```
SELECT customer_id, COUNT(*) AS delivered_orders
FROM orders
WHERE status = 'Delivered'        -- row filter
GROUP BY customer_id
HAVING COUNT(*) ⩾ 2;              -- group filter
```

Result: one row, customer 2 with 2 orders. The logical order of execution is FROM, WHERE, GROUP BY, HAVING, SELECT, ORDER BY, LIMIT. That is why a SELECT alias cannot be used in WHERE. (MySQL lets you use an alias in HAVING; PostgreSQL does not, so repeat the expression.)

Grouping by many columns and by date parts

Group by two columns and you get one row per unique pair. To group by month, cut the date down to its month first:

```
-- Works in both PostgreSQL and MySQL
SELECT EXTRACT(YEAR FROM order_date)  AS yr,
       EXTRACT(MONTH FROM order_date) AS mon,
       COUNT(*) AS orders, SUM(amount) AS revenue
FROM orders
GROUP BY EXTRACT(YEAR FROM order_date), EXTRACT(MONTH FROM order_date)
ORDER BY yr, mon;

-- PostgreSQL:  GROUP BY DATE_TRUNC('month', order_date)
-- MySQL:       GROUP BY DATE_FORMAT(order_date, '%Y-%m')
```

yr	mon	orders	revenue
2024	1	2	1700
2024	2	3	1100
2024	3	1	450

⚠️ **COMMON MISTAKE**

Grouping by MONTH alone. January 2023 and January 2024 then fall into the same bucket and your "January sales" doubles. Always group by year and month together, or use DATE_TRUNC.

🎯 **INTERVIEW CORNER**

Q: Can you use WHERE COUNT(*) > 5? **A:** No. WHERE runs before groups exist, so aggregates are not allowed there. Use HAVING COUNT(*) > 5. Put plain row conditions in WHERE, because filtering early is also faster.

CHECK YOUR PROGRESS 2.2

1. **Which clause filters groups by an aggregate value?**
   Answer: HAVING.
2. **Why must you add ORDER BY to a GROUP BY query?**
   Answer: GROUP BY does not guarantee any output order.

2.3 CASE WHEN and Conditional Aggregation

CASE is SQL's IF-ELSE. It checks conditions top to bottom and returns the first match. Without ELSE, unmatched rows get NULL. Use it to create labels and buckets from the `employees` table:

```
SELECT name, salary,
       CASE
         WHEN salary IS NULL  THEN 'Not set'
         WHEN salary ⩾ 80000 THEN 'High'
         WHEN salary ⩾ 50000 THEN 'Mid'
         ELSE 'Low'
       END AS band
FROM employees;
```

Notice the NULL check comes first. `NULL ⩾ 50000` is unknown, not true, so a NULL salary would quietly fall into ELSE and be called 'Low'.

You can GROUP BY the same CASE expression to count people per band (see Report 5).

Pivot: rows into columns

Put CASE inside SUM or COUNT and you get one column per category. This is called **conditional aggregation**, and it is the most loved interview trick in this chapter.

```
SELECT EXTRACT(MONTH FROM order_date) AS mon,
       SUM(CASE WHEN status = 'Delivered' THEN 1 ELSE 0 END) AS delivered,
       SUM(CASE WHEN status = 'Cancelled' THEN 1 ELSE 0 END) AS cancelled,
       SUM(CASE WHEN status = 'Delivered' THEN amount ELSE 0 END) AS delivered_value
FROM orders
GROUP BY EXTRACT(MONTH FROM order_date)
ORDER BY mon;
```

mon	delivered	cancelled	delivered_value
1	2	0	1700
2	1	1	800
3	0	1	0

Shortcuts: `COUNT(CASE WHEN status = 'Delivered' THEN 1 END)` works too, since COUNT skips the NULL from the missing ELSE. In MySQL a comparison returns 1 or 0, so `SUM(status = 'Delivered')` works there (not in PostgreSQL).

> 💡 **DO YOU KNOW?**
>
> PostgreSQL has a neat standard SQL clause: `COUNT(*) FILTER (WHERE status = 'Delivered')`. MySQL does not support FILTER, so the SUM(CASE ...) form is the portable answer.

> ✅ **CHECK YOUR PROGRESS 2.3**
>
> 1. **What does CASE return when no WHEN matches and there is no ELSE?**
>
>   **Answer:** NULL.
>
> 2. **What is conditional aggregation used for?**
>
>   **Answer:** To pivot categories into separate columns, e.g. SUM(CASE WHEN status = 'Cancelled' THEN 1 ELSE 0 END).

2.4 COALESCE, NULLIF, String and Date Functions

COALESCE(a, b, c) returns the first non-NULL argument. `COALESCE(amount, 0)` turns a missing amount into 0. **NULLIF(a, b)** returns NULL if a equals b, otherwise a. Together they give **safe division**:

```
SELECT 100.0 * SUM(CASE WHEN status = 'Cancelled' THEN 1 ELSE 0 END)
            / NULLIF(COUNT(*), 0) AS cancel_pct
FROM orders;
```

If COUNT(*) is 0, NULLIF turns it into NULL, and anything divided by NULL is NULL, so no crash. PostgreSQL throws "division by zero"; MySQL returns NULL with a warning. The `100.0` matters too: in PostgreSQL, integer / integer drops the decimals (7 / 2 = 3). MySQL gives 3.5000.

> 😄 **FUNNY SIDE**
>
> A report once showed a 0% cancellation rate for six months. Nobody had cancelled? No. Integer division had rounded 3 / 40 down to 0. The fix was one ".0". The intern got the credit.

Functions you will use in reports

Need	PostgreSQL	MySQL
Join text	`first || ' ' || last` or CONCAT	CONCAT(first, ' ', last)
Case, spaces	UPPER, LOWER, TRIM (both)	
Length in characters	LENGTH(s)	CHAR_LENGTH(s) (LENGTH counts bytes)
Part of text	SUBSTRING(s, 1, 3), LEFT(s, 3) (both)	
List in one cell	STRING_AGG(name, ', ')	GROUP_CONCAT(name SEPARATOR ', ')
Today	CURRENT_DATE (both)	
30 days ago	CURRENT_DATE - INTERVAL '30 days'	CURRENT_DATE - INTERVAL 30 DAY
Days between	d2 - d1 (dates)	DATEDIFF(d2, d1)
Year-month label	TO_CHAR(d, 'YYYY-MM')	DATE_FORMAT(d, '%Y-%m')

Need	PostgreSQL	MySQL
Round	ROUND(x, 2) (PostgreSQL needs x to be numeric)	

Careful: in MySQL, CONCAT returns NULL if any argument is NULL; PostgreSQL's CONCAT treats NULL as empty. Wrap risky columns in COALESCE. In MySQL, `||` means OR by default, not concatenation.

CHECK YOUR PROGRESS 2.4

1. **What does NULLIF(0, 0) return?**
   Answer: NULL.
2. **Write the PostgreSQL and MySQL expressions for "date 7 days before today".**
   Answer: CURRENT_DATE - INTERVAL '7 days' and CURRENT_DATE - INTERVAL 7 DAY.

2.5 Eight Worked Report Queries

These patterns cover most "write a query" rounds. Report 1 is PostgreSQL; the MySQL change is noted.

Report 1. Monthly revenue (delivered only)

```
SELECT TO_CHAR(order_date, 'YYYY-MM') AS month,   -- MySQL: DATE_FORMAT(order_date, '%Y-%m')
       COALESCE(SUM(amount), 0) AS revenue
FROM orders
WHERE status = 'Delivered'
GROUP BY TO_CHAR(order_date, 'YYYY-MM')
ORDER BY month;
```

Result: 2024-01 = 1700, 2024-02 = 800. March has no delivered order, so it simply does not appear.

Report 2. Department headcount and average salary, only departments with 2+ people

```
SELECT dept_id, COUNT(*) AS headcount,
       ROUND(AVG(salary), 0) AS avg_salary
FROM employees
GROUP BY dept_id
HAVING COUNT(*) ⩾ 2
ORDER BY headcount DESC;
```

dept_id	headcount	avg_salary
20	3	41500
10	2	75000

Dept 20 has three people but one NULL salary, so the average is (45000 + 38000) / 2.

Report 3. Repeat customers

```
SELECT customer_id, COUNT(*) AS orders
FROM orders
GROUP BY customer_id
HAVING COUNT(*) > 1;
```

Result: customers 1 and 2, with 2 orders each.

Report 4. Cancellation rate per customer

```
SELECT customer_id,
       ROUND(100.0 * SUM(CASE WHEN status = 'Cancelled' THEN 1 ELSE 0 END)
             / NULLIF(COUNT(*), 0), 1) AS cancel_pct
FROM orders
GROUP BY customer_id
ORDER BY cancel_pct DESC;
```

Result: customer 4 = 100.0, customer 1 = 50.0, customers 2 and 3 = 0.0.

Report 5. Employees per salary band

```
SELECT CASE WHEN salary IS NULL  THEN 'Not set'
            WHEN salary ⩾ 80000 THEN 'High'
            WHEN salary ⩾ 50000 THEN 'Mid'
            ELSE 'Low' END AS band,
       COUNT(*) AS employees
FROM employees
GROUP BY 1          -- position of the first SELECT column
ORDER BY employees DESC;
```

band	employees
Mid	2
Low	2
High	1
Not set	1

(Sample employees: salaries 90000, 60000, 45000, 38000, NULL, 52000. Ties may come in any order.)

Report 6. Daily active users

```
SELECT login_date, COUNT(DISTINCT user_id) AS dau
FROM logins
GROUP BY login_date
ORDER BY login_date;
```

DISTINCT matters: a user who logs in five times a day is still one active user.

Report 7. Hires per year, pivoted by department

```
SELECT EXTRACT(YEAR FROM hire_date) AS yr,
       COUNT(CASE WHEN dept_id = 10 THEN 1 END) AS dept_10,
       COUNT(CASE WHEN dept_id = 20 THEN 1 END) AS dept_20,
       COUNT(CASE WHEN dept_id = 30 THEN 1 END) AS dept_30
FROM employees
GROUP BY EXTRACT(YEAR FROM hire_date)
ORDER BY yr;
```

Report 8. Salary spread per department

```
SELECT dept_id, MIN(salary) AS lowest, MAX(salary) AS highest,
       MAX(salary) - MIN(salary) AS spread
FROM employees
GROUP BY dept_id;
```

Result: dept 10 spread 30000, dept 20 spread 7000, dept 30 spread 0.

🔑 **KEY POINT**

Most report questions are one of three shapes: "X per Y" (GROUP BY), "only Y where total is big" (HAVING), and "X split by category as columns" (SUM(CASE ...)). Spot the shape first, then type.

🎯 **INTERVIEW CORNER**

Q: COUNT(*) or COUNT(1), which is faster? **A:** They are the same in PostgreSQL and MySQL; both count rows. The real difference is with COUNT(col), which skips NULLs and can give a different answer.

🖍️ **TRY IT**

Using the orders table, write one query that shows, per month, total orders, delivered orders, and the delivery percentage rounded to 1 decimal. Check: January should be 100.0.

✅ **CHECK YOUR PROGRESS 2.5**

1. **Why does Report 6 use COUNT(DISTINCT user_id) instead of COUNT(*)?**
   Answer: One user can have many login rows a day; we want unique users.
2. **In Report 1, why is March missing?**
   Answer: WHERE removed its only (cancelled) order before grouping, so no March group exists.

What Have We Learnt?

- Aggregates turn many rows into one; all except COUNT(*) ignore NULL.
- COUNT(*) counts rows, COUNT(col) non-NULL values, COUNT(DISTINCT col) unique non-NULL values.
- Every SELECT column must be grouped or aggregated; always add ORDER BY.
- WHERE filters rows before grouping; HAVING filters groups after.
- CASE WHEN creates labels and buckets; SUM(CASE ...) pivots rows into columns.
- COALESCE fills NULLs; x / NULLIF(y, 0) is safe division; use 100.0 to avoid integer division.

Practice

A. Multiple Choice Questions

1. COUNT(amount) counts:
   a. All rows
   b. Rows where amount is not NULL
   c. Unique amounts
   d. Rows where amount is NULL
2. To keep only departments with average salary above 50000, use:
   a. WHERE AVG(salary) > 50000
   b. ORDER BY AVG(salary)
   c. HAVING AVG(salary) > 50000
   d. GROUP BY salary > 50000
3. The safe way to compute a / b when b may be 0 is:
   a. a / COALESCE(b, 0)
   b. a / NULLIF(b, 0)
   c. COALESCE(a / b)
   d. a / (b + 0)
4. AVG of 10, NULL, 20 is:
   a. 10
   b. NULL
   c. 15
   d. 30

B. Practical Tasks

1. List each customer's total delivered spend, showing only customers who spent more than 1000.
2. From logins, show each user's number of logins in January 2024 and February 2024 as two columns.
3. From customers, count signups per city per year, latest year first.

Answer Key: Chapter 2

A. MCQ

1. (b) COUNT(col) skips NULLs.
2. (c) Aggregate conditions go in HAVING.
3. (b) NULLIF turns 0 into NULL, so the result is NULL, not an error.
4. (c) NULL is ignored: (10 + 20) / 2 = 15.

B. Practical Tasks

1.
```
SELECT customer_id, SUM(amount) AS spend
FROM orders
WHERE status = 'Delivered'
GROUP BY customer_id
HAVING SUM(amount) > 1000;
```

Sample data: customer 2 with 2000.

2.
```
SELECT user_id,
       SUM(CASE WHEN login_date ⩾ DATE '2024-01-01' AND login_date < DATE '2024-02-
01' THEN 1 ELSE 0 END) AS jan,
       SUM(CASE WHEN login_date ⩾ DATE '2024-02-01' AND login_date < DATE '2024-03-
01' THEN 1 ELSE 0 END) AS feb
FROM logins
GROUP BY user_id;
```

3.
```
SELECT city, EXTRACT(YEAR FROM signup_date) AS yr, COUNT(*) AS signups
FROM customers
GROUP BY city, EXTRACT(YEAR FROM signup_date)
ORDER BY yr DESC, city;
```